JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **47,** 39–57 (1997)
ARTICLE NO. PC971394

Competitive Coevolutionary Multi-Agent Systems: The Application to Mapping and Scheduling Problems

Franciszek Seredynski[1]

Institute of Computer Science, Polish Academy of Sciences, Ordona 21, 01-237 Warsaw, Poland

A new paradigm for a parallel and distributed evolutionary computation is proposed in this paper. The main idea of the proposed approach is based on considering a given system as a multi-agent system with game-theoretic models of interaction between players. For this purpose a model of noncooperative *N*-person games with limited interaction is considered. Each player in the game has a payoff function and a set of actions. While players compete to maximize their payoffs, we are interested in the global behavior of the team of players, measured by the average payoff received by the team. To evolve a global behavior in the system, we propose three distributed schemes with evaluation of only local fitness functions. The first scheme uses ε-learning automata and is compared with two coevolutionary schemes, which we call loosely coupled genetic algorithms and loosely coupled classifier systems, respectively. We present simulation results which indicate that the global behavior in the systems emerges and is achieved in particular by only a local cooperation between players acting without global information about the system. The models of multi-agent systems are applied to develop parallel and distributed algorithms of dynamic mapping and scheduling tasks in parallel computers.

1. INTRODUCTION

One of the widely accepted assumptions concerning the general model of real life systems is to consider them as a collection of individuals acting selfishly and interacting to fulfill their own goals. The immediate consequence of this assumption is conflict between individuals and the need for some cooperation to resolve the conflict. For this reason, game theory [22] modeling conflict and cooperation has been widely accepted as a useful tool to study the behavior of complex systems.

A game-theoretic model which has received much attention of late is the prisoner's dilemma. This two-player, noncooperative game has been explored to yield insight into the conditions of cooperating and defecting between players. Evolutionary computation methods using genetic algorithms [4] or evolutionary programming [10] have been successfully used to discover Tit for Tat strategy or better strategies of behavior. One of the distinctive features of the prisoner's dilemma is a focus on a player's individual goal. Cooperation in this model is desired and observed, but it is not the ultimate aim of a game.

In the area of distributed artificial intelligence (DAI), game-theoretic models are also the subject of current research [16]. Agents of a multi-agent system in such models are considered as players working toward their own selfish goals and taking part in an *N*-person game. In many DAI applications (e.g., in engineering), a global behavior of the multi-agent systems, rather than simply a fulfillment of players' individual goals, is expected. Competing players in such systems should act as a decision group choosing their actions to realize a global goal. One of the problems which must be addressed here is the problem of incorporating the global goals of the multi-agent system into the local interests of all agents. The obvious conflict between individual and global goals is not solved, to our knowledge, on the grounds of game theory literature concerning social, political, or economic phenomena.

This paper is application driven. We are interested in models of *N*-person games and their potential application in selected problems in the area of parallel and distributed computing [11]. In particular, mapping and scheduling tasks [5, 9] of parallel programs in massively parallel and distributed computers belong to the central questions to be solved to efficiently use their computational power.

The paper is organized as follows. The following section presents the model of *N*-person games with limited interaction. We discuss the model from the position of the collective behavior of players and the conditions of cooperation between them, providing a maximal payoff for the team of players in a static game. In Section 3, we propose three parallel and distributed schemes to implement iterated *N*-person games. These are games of ε-learning automata, loosely coupled genetic algorithms, and loosely coupled classifier systems. Section 4 presents an application of the multi-agent systems to problems of dynamic mapping and scheduling tasks in parallel computers. The last section contains conclusions.

[1]E-mail: sered@ipipan.waw.pl.

0743-7315/97 $25.00

TABLE I
The Payoff Function Used in the Prisoner's Dilemma

		Player B: **C**	Player B: **D**
Player A	**C**	$\gamma_1 = 3,\ \gamma_1 = 3$	$\gamma_2 = 0,\ \gamma_3 = 5$
	D	$\gamma_3 = 5,\ \gamma_2 = 0$	$\gamma_4 = 1,\ \gamma_4 = 1$

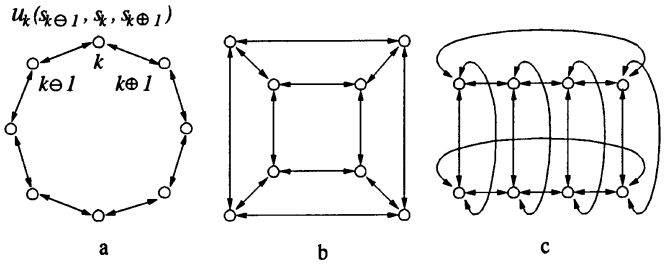

FIG. 1. Interaction graph of a game on a ring (a), a cube (b), and a torus (c).

2. *N*-PERSON GAMES WITH LIMITED INTERACTION

2.1. Prisoner's Dilemma Background

The typical two person *prisoner's dilemma* is described by a matrix shown in Table I [4, 10].

It is assumed that each player has two alternative actions: **C**, cooperate and **D**, defect. The purpose of each player is to maximize personal profit in the game. The game is conducted as a sequence of trials. During each trial, both players must choose independently one of their actions. After each trial, the players receive a payoff whose value is defined by one of four situations: (**C**, **C**), player A and player B cooperate and both receive the payoff $\gamma_1 = 3$; (**D**, **D**), both players defect and receive the payoff $\gamma_4 = 1$; (**C**, **D**), player A cooperates while player B defects and the players obtain the payoff $\gamma_2 = 0$ and $\gamma_3 = 5$, respectively; (**D**, **C**), player A defects and player B cooperates, and they obtain the payoff $\gamma_3 = 5$ and $\gamma_2 = 0$, respectively. The values of the payoff defining the game are the subject of some constraints [10]. To cooperate with the opponent player and to receive the payoff $\gamma_1 = 3$ or to defect obtaining the higher payoff $\gamma_3 = 5$ while the opponent continues to cooperate is the main question of current studies [38] concerning prisoner's dilemma model.

In our model of *N*-person games with limited interaction described below we use the basic notation from the prisoner's dilemma; however, the structure of the payoff function in the game considered below is different, in general. Our approach to study the game also differs from the prisoner's dilemma in stressing the necessity to realize the global goal of the system.

2.2. Games with Limited Interaction

We consider a finite game **G** represented by a set **N** of N players, $\mathbf{N} = \{0, 1, \ldots, N-1\}$; a set S_k of actions for each player $k \in \mathbf{N}$; and a payoff function $u_k(s_k, s_{k1}, s_{k2}, \ldots, s_{kn_k})$ which depends only on the actions of a limited number of players—on its own action s_k and the actions of its n_k neighbors in the game. Such a model, termed a *game with limited interaction*, had been primarily considered from positions of learning automata games [26, 32, 34].

The game with limited interaction can be represented by an oriented graph $G = \langle V, E \rangle$ called an *interaction graph*. V is the set of nodes corresponding to the set of players while set E represents the pattern of interaction between players: arcs incoming to the kth node define players whose actions influence the payoff of player k, and arcs outgoing from the kth node define players whose payoff depends on the actions of player k. Each player in a game has his own payoff function which is different in general.

In the paper we consider a class of games with limited interaction characterized by a regular interaction graph ($n_k = r$, where r is the degree of the interaction graph) and called homogeneous games. In such games, the payoff function is the same for all players. Figure 1 shows the interaction graphs of three examples of homogeneous games, each with a number of players equal to $N = 8$.

The simplest homogeneous game with limited interaction is a game on a ring (Fig. 1a). For the game on a ring, which will be the subject of our study in this paper, we can simplify notation of the payoff function to read:

$$u_k(s_k, s_{k1}, s_{k2}, \ldots, s_{kn_k}) = u_k(s_{k\ominus 1}, s_k, s_{k\oplus 1}), \qquad (1)$$

where $\ominus$ and $\oplus$ denote subtraction and addition modulo N, respectively. A payoff function of any player in a game on the ring depends on its actions and on actions of its two neighbors $k \ominus 1$ and $k \oplus 1$. Assuming that the set S_k of actions for each player is limited to the set {**D**, **C**}, the payoff function has eight entries and Table II shows two examples u_k^1 and u_k^2 of a payoff function for a game on a ring.

TABLE II
Two Payoff Functions u_k^1 and u_k^2 Used in a Game on a Ring

	$s_{k\ominus 1}$	s_k	$s_{k\oplus 1}$	$u_k^1(s_{k\ominus 1}, s_k, s_{k\oplus 1})$	$u_k^2(s_{k\ominus 1}, s_k, s_{k\oplus 1})$
0	**D**	**D**	**D**	10	10
1	**D**	**D**	**C**	0	0
2	**D**	**C**	**D**	0	0
3	**D**	**C**	**C**	0	0
4	**C**	**D**	**D**	0	0
5	**C**	**D**	**C**	50	30
6	**C**	**C**	**D**	0	0
7	**C**	**C**	**C**	30	50

It is supposed that each player acts independently in the game and selects actions to maximize his payoff. If players play the game defined by the payoff function u_k^1, and player k and his neighbors $k \ominus 1$ and $k \oplus 1$ all select action **D** in a trial, his payoff will be defined by a sequence of actions (**D**, **D**, **D**) and is equal to 10 (entry 0 of Table II). If player k selects action **C** while both neighbors select action **D**, player k will receive the payoff equal to 0 (entry 2 of Table II). The remaining entries of Table II are self-explaining.

The most widely used solution concept for noncooperative games is a *Nash equilibrium point* [22]. A Nash point is an N-tuple of actions, one for each player, such that anyone who deviates from it unilaterally cannot possibly improve his expected payoff. If s_k denotes an action of the kth player, then a Nash equilibrium point is an N-tuple $(s_1^*, s_2^*, \ldots, s_k^*, \ldots, s_N^*)$ such that

$$u_k(s_0^*, s_1^*, \ldots, s_k^*, \ldots, s_{N-1}^*) \geq u_k(s_0^*, s_1^*, \ldots, s_k, \ldots, s_{N-1}^*) \quad (2)$$

for $s_k \neq s_k^*$ and $k = 0, 1, \ldots, N-1$. A Nash equilibrium point will define payoffs of all players in the game. However, we are not interested in the payoff of a given player, but in some global measure of the payoff received by the team of players. This measure can be, e.g., the average payoff $\overline{u}(s)$ received by the team as a result of their combined actions, $s = (s_0, s_1, \ldots, s_{N-1})$, i.e.,

$$\overline{u}(s) = \left(\sum_{k=0}^{N-1} u_k(s)\right) \Big/ N, \quad (3)$$

and it will be our global criterion to evaluate the behavior of the players in the game. The question which arises immediately concerns the value of (3) in a Nash point. Unfortunately, this value can be very low.

Analyzing all possible actions' combinations in the game and evaluating their prices, i.e., a value $\overline{u}(s)$, we can find actions' combinations characterized by a maximal price and we can call them *maximal price points*. Maximal price points are actions' combinations which maximize the global criterion (3), but they can be reached by players only if they are Nash points. A maximal price point usually is not a Nash point and the question which must be solved is how to convert a maximal price point (points) into a Nash point (points).

2.3. Exchange Processes in Games

It is useful from the point of view of a decentralization of behavior of players in games with limited interactions to introduce the notion of an *exchange process* [26, 33].

The exchange process in a game **G** is a procedure of redistribution of payoffs between players accoridng to the following rules:

(a) the exchange process is given by an oriented graph,
(b) for each kth vertex of the graph, $p_k + 1$ is the number of arcs going out of it and $q_k + 1$ is the number of arcs coming into it,
(c) u_k is a payoff of a player k (which corresponds to the kth vertex of the graph) in some actions' combination of a game,
(d) each player sends on outgoing arcs a part of its payoff equal to $u_k/(p_k + 1)$,
(e) each player receives on an incoming arc from each player l a part of its payoff equal to $u_l/(p_l + 1)$.

Players taking part in a game with an exchange process share their payoffs. The exchange process results in a transformation of a payoff function of a game **G** into a new payoff of a new game. Using the idea of an exchange process introducing sharing payoffs in a game, we will consider the following schemes of a game:

- game **G**: *no cooperation* in a game; i.e., a payoff of a player k is defined by u_k,
- game $\mathbf{G}^*$: *local cooperation*—a player k shares his payoff with his neighbors in a game; i.e., his payoff is transformed as:

$$u_k \longrightarrow w_k = \frac{u_k + \sum_{l \in N_k} u_l}{\max_{l \in \mathbf{N}} n_l + 1}, \quad (4)$$

where $N_k \subset \mathbf{N}$ is a set of neighbors of a player k, u_l is a payoff of a player's k neighbor l, and $\max_{l \in \mathbf{N}} n_l$ denotes a maximal number of neighbors in a game,

- game $\mathbf{G}^{**}$: *global cooperation*—sharing a payoff received by a player k by all players participating in a game; i.e., his payoff u_k is transformed into a new payoff w_k in the following way:

$$u_k \longrightarrow w_k = \overline{u}(s). \quad (5)$$

A *conjugate exchange process* in a game with limited interaction is an exchange process whose graph coincides with an interaction graph of the game. The conjugate exchange process in a homogeneous game with limited interaction corresponds to the organization of local coalitions with neighbors in the game. In fact each player in the game takes part simultaneously in $n_k = r$ coalitions, where r is the degree of the interaction graph. Figure 2a shows the graph of a conjugate exchange process for the game on a ring from Fig. 1a.

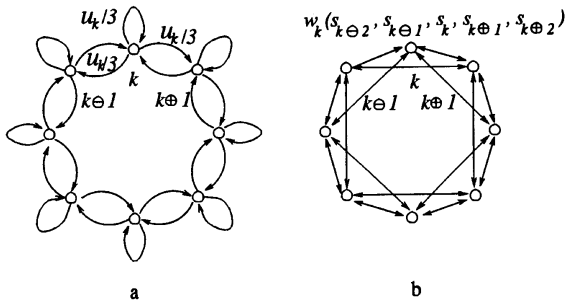

FIG. 2. Game on a ring: graph of a conjugate exchange process (a), and interaction graph of the game transformed by the conjugate exchange process (b).

A game with a conjugate exchange process corresponds to a class of games $\mathbf{G}^*$ with a local cooperation. It can be shown [26, 33] that introducing a conjugate exchange process into a homogeneous game **G**, i.e., converting it into a game $\mathbf{G}^*$, results in the transformation of maximal price points of the game **G** into Nash points of the game $\mathbf{G}^*$. It means that maximal price points of the game **G** become Nash points of the game $\mathbf{G}^*$ in the result of only local cooperation between neighbors' players.

For the game on a ring from Fig. 1a, the conjugated exchange process transforms the payoff u_k of the player k into a new payoff w_k in the following way:

$$u_k \longrightarrow w_k = (u_{k\ominus 1} + u_k + u_{k\oplus 1})/3. \tag{6}$$

It also results in the transformation of the interaction graph of the game as shown in Fig. 2b.

A *total exchange process* in a game is an exchange process described by a complete graph. A game with a total exchange process corresponds to a class of games $\mathbf{G}^{**}$ with a global cooperation. In such a game each player takes part simultaneously in $N - 1$ coalitions. It also can be shown that maximal price points in such a game are transformed into Nash points. Achieving a global goal (i.e., playing a maximal price point converted into a Nash point, providing a maximum of (3)) by players in a game $\mathbf{G}^{**}$ needs, in opposition to a game $\mathbf{G}^*$, a cooperation of all players participating in a game.

3. ITERATED GAMES WITH LIMITED INTERACTION

3.1. Analysis of Payoff Functions u_k^1 and u_k^2

We are interested in the behavior of players in dynamic games and therefore we will study iterated N-players games with limited interactions. An *iterated game* consists of a number T_g of single games $s(t) = (s_0(t), s_1(t), \ldots, s_{N-1}(t))$ played in subsequent moments of time $t = 0, 1, \ldots, T_g - 1$, with a value T_g unknown for players. In a single game played at the moment t each player autonomously selects an action to match an observed state $x_k(t)$ of a game environment. Observed by a player k state $x_k(t)$ of the environment is formed by him and his neighbors' actions played in a previous moment of time. Rewards defined by a payoff function are transferred to players directly or after their redistribution, if an exchange process is used.

We will study iterated games by observation of the global criterion (3) and we will particularly be interested in the following situations:

- there exist maximal price points in a game, which are not Nash points,
- taking into account the above condition, but with players creating coalitions defined by the conjugate or total exchange processes.

Let us first analyze the payoff function u_k^1 from Table II for the game **G** on the ring (Fig. 1a), assuming $N = 8$ players participating in the game. The game has the maximal price point s_{mp} = (**C, C, C, C, C, C, C, C**). Each player taking part in the game defined by the vector of actions s_{mp} obtains a payoff equal to $u^1_{k,7}$(**C, C, C**) = 30. The average payoff $\overline{u}_k^1$ received by the team of players achieves the maximal value equal to 30. However, the actions' combination s_{mp} is not a Nash point because it is reasonable for each player to change his action **C** → **D** and obtain a higher payoff equal to $u^1_{k,5}$(**C, D, C**) = 50. The game has several Nash points. For one of them, Nash point s_N = (**D, D, D, D, D, D, D, D**), the inequality (2) is strict for all values of k. One can see that in this game there is no reason for any player to change his action **D** → **C** and obtain instead of the payoff $u^1_{k,0}$(**D, D, D**) = 10, a lower payoff equal to $u^1_{k,2}$(**D, C, D**) = 0. We can therefore expect that in a game **G** players will not achieve the maximal price point but will rather play this Nash game.

In a game $\mathbf{G}^*$ the maximal price point is transformed into a Nash point. One can see that in such a game, while playing the maximal price point game, there is no reason for any player to change his action **C** → **D**. In the case of changing, his new payoff calculated according to (6) will be equal to w_k(**C, C, D, C, C**) = ($u_{k\ominus 1}$(**C, C, D**) + u_k((**C, D, C**) + $u_{k\oplus 1}$(**D, C, C**))/3 = (0 + 50 + 0)/3 = 16.66. This new payoff is lower than the payoff equal to 30, corresponding to the maximal price point.

A similar situation is in the game $\mathbf{G}^{**}$. A new payoff of any player, e.g., player 3, changing his action **D** → **C** while others continue playing the game corresponding to the maximal price point game, will result in a new worse payoff calculated according to (5): w_k(**C, C, C, D, C, C, C, C**) = (u_0(**C, C, C**) + u_1(**C, C, C**) + u_2(**C, C, D**) + u_3(**C, D, C**) + u_4(**D, C, C**) + u_5(**C, C, C**) + u_6(**C, C, C**) + u_7(**C, C, C**)/8 = (30 + 30 + 0 + 50 + 0 + 30 + 30 + 30)/8 = 25. We can therefore expect that in both games $\mathbf{G}^*$ and $\mathbf{G}^{**}$ rational players will find and play the maximal price point game.

It is worth noticing that the example of the payoff function u_k^2 from Table II does not contain in its structure an entry corresponding to the prisoner's dilemma situation. It is easy to see that the maximal price point s_{mp} = (**C, C, C, C, C, C, C, C**) is at the same time a Nash point.

Introducing into a game **G** with a payoff function such as u_k^1 an exchange process converts in fact a structure of a payoff function of the game into a structure similar to that shown in the payoff function u_k^2. For this reason we will study below iterated games with the payoff function u_k^1. The number of players participating in all games is equal to $N = 16$. For each such game, the number of possible game solutions (combinations of actions) is equal to 2^{16}. All experiments have been conducted on a PC system.

3.2. Games of ε-learning Automata

Behaviors of learning automata [21, 32] in N-person games have been studied from positions of collective behavior of automata [24, 32, 34]. A typical assumption in such games is that a payoff received by an automaton is a random value defined by a stochastic environment. Also automata taking part in games are usually stochastic. We are going to

apply automata models to a deterministic variant of mapping and scheduling problems. Environments defined by these problems are deterministic, and therefore it is reasonable to use *deterministic* learning automata in such games.

We will use deterministic *ε-learning automata* [34] as players in the games with limited interaction. An ε-learning automaton has d actions and acts in a deterministic environment $c = (c_1, c_2, \ldots, c_d)$, where c_i stands for a reward obtained for its action s_i. Whenever an automaton generates an action, the environment sends it a payoff in a deterministic way. The objective of a reinforcement learning algorithm represented by ε-automaton is to maximize its payoff in an environment where the automaton operates. The automaton remembers its last h actions and corresponding payments. An ε-automaton chooses as the next action its best action from the last h games with the probability $(1 - \varepsilon)$ $(0 < \varepsilon < 1)$, and with the probability ε/d any of its d actions.

Figure 3 shows results of an experimental study of the iterated game of ε-learning automata. It shows the average payoff

$$\overline{U}(s) = \left(\sum_{t=0}^{T_g-1} \sum_{k=0}^{N-1} u_k(s(t)) \right) \Big/ (T_g * N) \qquad (7)$$

of a team of players obtained in the iterated game without cooperation, with a local cooperation, and with a global cooperation. Each point on the curves represents the mean of 30 runs of the game.

The figure shows the ability to produce an effective collective behavior by a team of learning automata in the iterated game with a local and global cooperation. In both types of the game the average payoff $\overline{U}(s)$ of the team of players after 10,000 games approaches a value nearly equal to 30, defined by the maximal price point of the game. In the game without a cooperation the system quickly finds and plays mostly a game corresponding to the Nash point, with a low average payoff $\overline{U}(s)$ equal to 10. The ability of convergence of automata to a global optimum in a game with a local and global cooperation, under constant values ε and h, depends on a number of played games. The number of necessary games to achieve a global optimum is large (more than 10,000 for the conducted experiment), and therefore we are interested in searching for new, more effective schemes implementing the game. In the next sections we consider an iterated game as a *coevolutionary system*, applying to it genetic algorithms (GA) [13, 19] methodology.

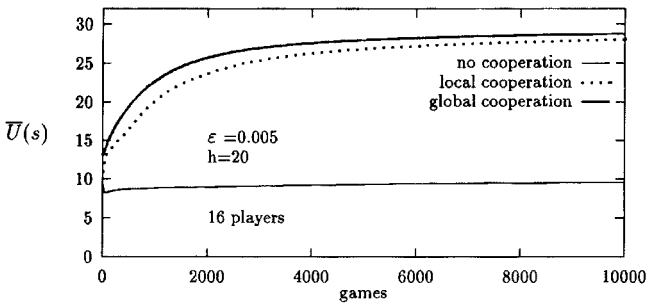

FIG. 3. The average payoff received by the team of ε-learning automata.

3.3. Loosely Coupled Genetic Algorithms

Attempting to apply GA methodology to iterated games with limited interaction we face two difficulties: (a) how to create a GA model of a player which acts concurrently to maximize its payoff, and at the same time (b) how in a dynamic process of the game to search for a solution of the game according to the global criterion (3), i.e., to search for a maximal price point. If we base ourselves on the positions of traditional GA, we should emphasize the second issue, but in this case we lose the concurrent nature of the game. If we emphasize the concurrent behavior of players, it seems that most sequential GA [13, 19] as well as parallel GA [20, 35] of both *island* and *diffusion* models are not suitable for our purpose.

It is typical of these models to evaluate a *global fitness* of an individual representing a global solution of a problem, despite its belonging to a global population or a subpopulation. For this reason, we call this class of GA *tightly coupled* GA. For our purpose, we need an evolutionary system with coevolving subpopulations representing behavior of players, evaluating only their local fitness functions and, at the same time, we expect from such a model a global behavior, in the sense of searching for a global optimum expressed by some composition of local functions.

The need of an extension of the traditional GA into the direction of coevolutionary systems has been lately recognized. While a *cooperative coevolutionary approach* (tightly coupled GA) to function optimization has been proposed [23], *loosely coupled* GA (LCGA) with evaluation of *local functions* and *competition* between subpopulations has been suggested [27] to support the above described game-theoretic model of computation, suitable for distributed decision-making.

The idea of LCGA implementing the game on a ring from Fig. 1a is shown in Fig. 4. The LCGA can be specified in the following way:

1. For each player create an initial subpopulation of its actions
   - create randomly for each player an initial population of size n of player actions taking values from the set S_k (see Section 2.2) of its actions; Fig. 4 shows the initial subpopulations of the size $n = 4$ of actions for players $k \ominus 1$, k, and $k \oplus 1$, respectively; the value of n defines a game horizon for a player; actions predefined in a subpopulation of a given player will be used in subsequent n iterated games (see Fig. 5).
2. Play a single game
   - in a discrete moment of time each player selects randomly one action from the set of actions predefined in its subpopulation and not used until now and presents it to his neighbors in the game; shadowed actions in Fig. 4 show possible situations after some game: players $k \ominus 1$, k, and $k \oplus 1$ have selected actions **D**, **D**, and **C**, respectively; the chain of selected actions can be interpreted as a possible solution of

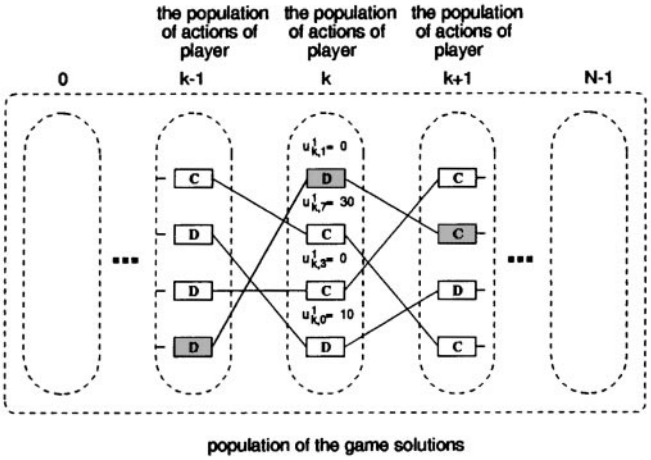

FIG. 4. Loosely coupled genetic algorithm implementing a game on a ring.

the game.

• calculate the output of the game: each player evaluates its local payoff u_k in the game; if a payoff function in the game is the function u_k^1 from Table II then player k obtains for action **D** (Fig. 4, shadowed box) the payoff $u_{k,1}$(**D, D, C**) = 0 ($u_{k,1}^1$: read, the entry 1 of the payoff function u_k^1).

• if the game with an exchange process is played, each player informs his partners in the game about his current payoff and calculates his modified payoff w_k.

3. Repeat Step 2 until n games are played.

4. Using GA operators create for each player a new subpopulation of his actions

• after playing n games each player knows the value of his payoff received for a given action from its subpopulation; chains of actions present actually found solutions of the game

• the values of the payoff are considered as values of a local fitness function defined during a given generation of GA; standard GA operators of selection (*S*), crossover (*Cr*) (this operator is not used here because actions are limited to two binary digits), and mutation (*M*) are applied locally to subpopulations of actions; these actions will be used by players in games played in the next game horizon.

5. Return to Step 2 until the termination condition is satisfied

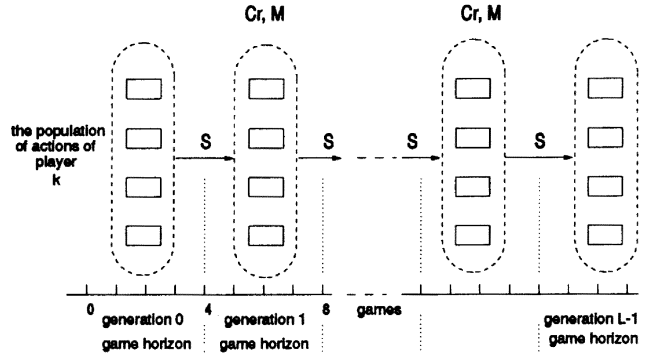

FIG. 5. Actions from evolving subpopulations of a player k are used in subsequent game horizons to play the game on a ring implemented with LCGA.

• if a given population evolved during L generations then the number of played games T_g can be defined as

$$T_g = n * L. \tag{8}$$

It is worth noticing that playing n games corresponding to a generation does not need in fact n units of time, but can be performed in 1 unit of time. A player predefines his own order of actions used in a game horizon and sends this order together with actions to his neighbors. We can think about parallel execution of n games corresponding to a players' subpopulation. It is significantly different from games of ε-learning automata, where taking a decision about the next action needs exchanging information about decisions in subsequent moments of time, after each single game.

Below we study the behavior of players in the game implemented with LCGA. In all experiments we observed the game during 100 generations. Figure 6 shows the average payoff

$$\overline{U}(s) = \left(\sum_{i=0}^{L-1} \sum_{j=1}^{n} \sum_{k=0}^{N-1} u_k(s) \right) \Big/ (n * L * N) \tag{9}$$

of a team of players, obtained in the iterated game without cooperation, with a local and a global cooperation, respectively. Each point of the curves represents the mean of 30 runs of the game.

The figure shows the average payoff received by the team of players as a function of a number of generations. One can see that in the game implemented with LCGA a global behavior evolves in the case of games with a local and global cooperation, similar to games of ε-automata. In the game without a cooperation players play mostly games corresponding to the Nash point. Behavior of players in games largely depends on the size n of a population of players' actions.

The common feature of the game with a local cooperation implemented either with ε-automata or LCGA is their ability of the global behavior which is realized in a fully distributed manner, without any knowledge about either the global optimization criterion or a number of players participating in the game. A single game under both implementations with a number of players equal to N, and a number of neighbors in the game equal to r, needs only $r * N$ interaction (sending messages) between players.

In the game with a global cooperation we can also observe a global behavior in both implementations of the game. While players act independently, a global cooperation between players is required, which needs $N * (N - 1)$ interactions between players in a single game. It is clear therefore that a game with a local cooperation is much more interesting from the point of view of potential applications.

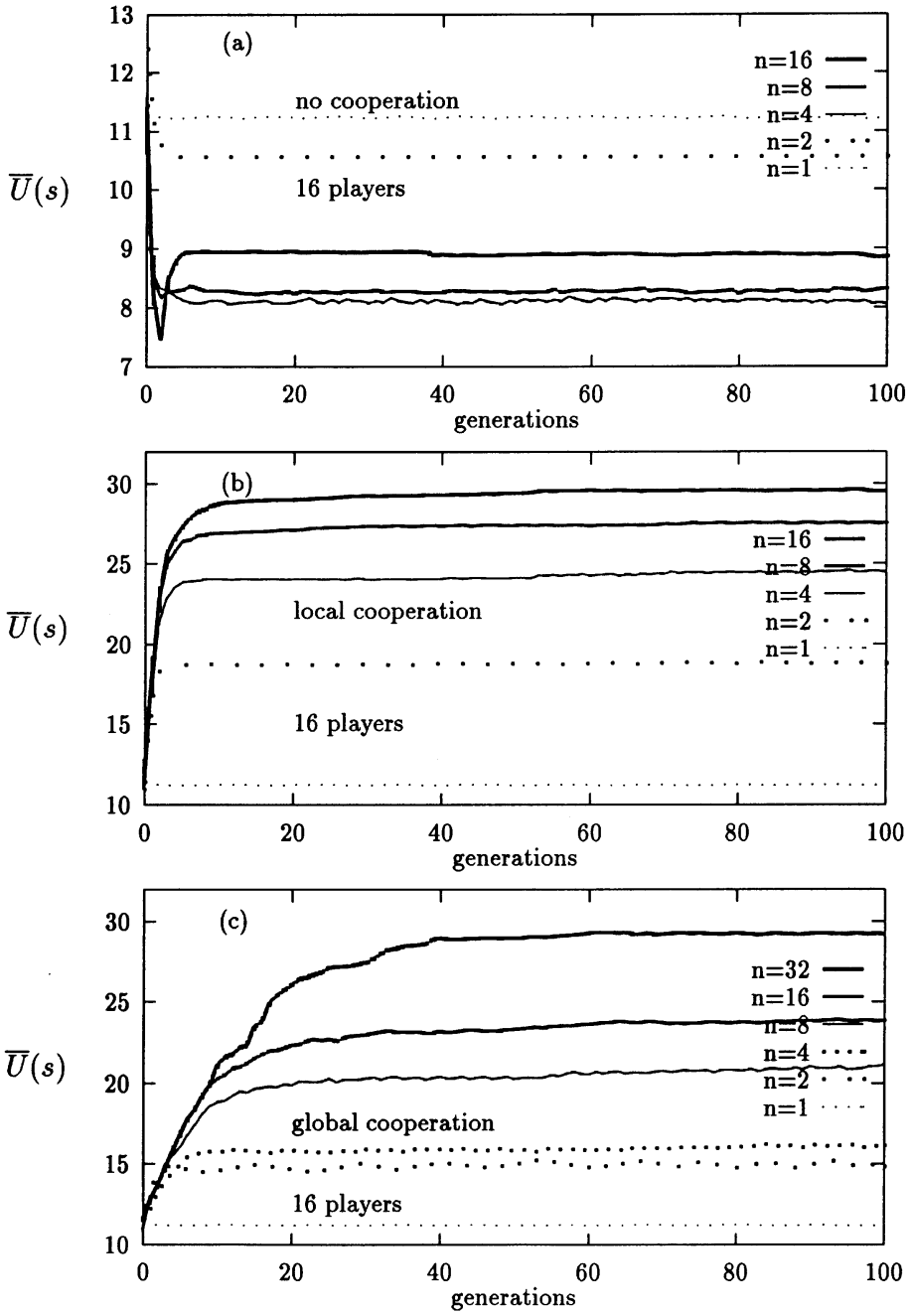

FIG. 6. The average payoff received by the team of players in the game implemented with LCGA without cooperation (a), with a local cooperation (b), and with a global cooperation (c).

One can see some similarity between the LCGA model and GA model based on a *niche formation* concept described by Goldberg [13]. In both models a transformation of a fitness function of strings is performed with the use of sharing mechanisms, which are different. While the GA model based on a niche formation concept works with a global population and a global fitness function evaluation (tightly coupled GA), the LCGA has to do with competing subpopulations and assigning to them locally defined functions.

The distinctive feature of games implemented with LCGA is their very fast speed of convergence. The number of iterated games played by a system implemented with LCGA can be calculated according to (8). In the case of a game with a local cooperation with $n = 16$, the number of generations needed for converging is equal to about 50 (see Fig. 6), which corresponds to a number of games $T_g = 16 * 50 = 800$, while for the same game with the use of ε-automata a required number of games is expressed in thousands of games (see Fig. 3).

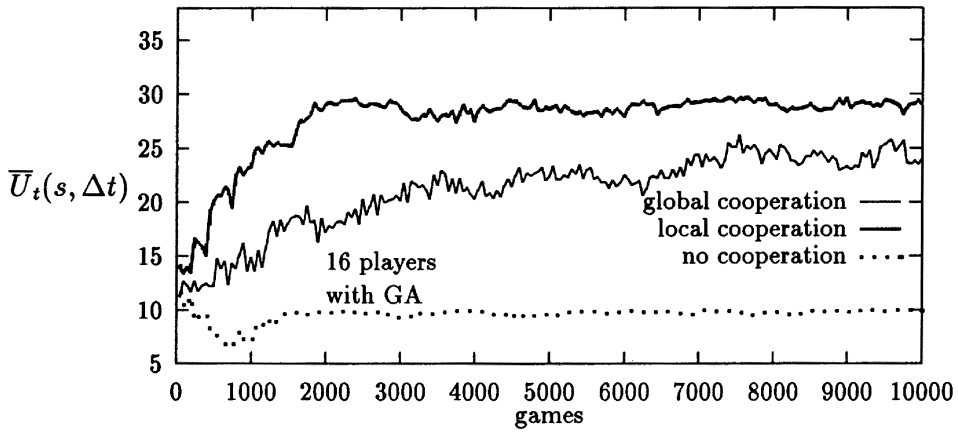

FIG. 7. The average payoff of classifier systems in a game on a ring without cooperation, with a local cooperation, and with a global cooperation.

3.4. Loosely Coupled Classifier Systems

In this section we study the behavior of a game-theoretic multi-agent system from Section 2 when each agent is a genetics-based machine learning system called a *classifier system* [13].

Classifier systems constitute the most popular approach to genetics-based machine learning. A learning classifier system (CS) maintains a population of decision rules, called *classifiers*, evaluated by using them to generate actions and observing received rewards defined by a payoff function and modified by periodically applying GA.

A classifier c is a condition–action pair

$$c = \langle \textbf{condition} \rangle : \langle \textbf{action} \rangle,$$

with the interpretation of the following decision rule: if a current observed state matches the **condition**, then execute the **action**. The action part of a classifier is an element of the set S_k of actions a player k in a game with limited interaction. The conditional part of a classifier of CS representing a player k contains his action and actions of his neighbors in the game, and additionally a *don't-care* symbol #.

A real-valued *strength* of a classifier is estimated in terms of rewards obtained according to a payoff function of a player k, using by the player the given classifier to generate an action. Action selection is implemented by a competition mechanism, where the winner is a classifier with the highest strength.

To modify classifier strengths the simplified *credit assignment* algorithm [13] was used. The algorithm consists of subtracting a tax of the winning classifier from its strength and then dividing equally the reward received after executing an action, among all classifiers matching the observed state.

To create new classifiers a standard GA is applied, with three basic genetic operators: selection, crossover, and mutation. Crossover is applied only to the conditional parts of classifiers. Mutation consists of altering with a small probability randomly selected condition elements or actions. GA is invoked periodically and each time it replaces some classifiers with new ones.

A number of experiments with the use of CS as players in the iterated game on a ring has been conducted. Figure 7 shows some results of experiments with 16 players participating in an iterated game defined by a payoff function from Table II and consisting of 10,000 games. The figure shows the average payoff $\overline{U}_t(s,\ \Delta t)$,

$$\overline{U}_t(s,\ \Delta t) = \left(\sum_{t'=\text{entier}(t/\Delta t)}^{\text{entier}(t/\Delta t)+\Delta t-1} \overline{u}(s,\ t') \right) \Big/ (\Delta t), \qquad (10)$$

of players in a single run of the game, received during each $\Delta t = 50$ games, in a game without cooperation, in a game with a local cooperation, and in a game with a global cooperation, respectively.

The behavior of CS in the game is similar to the behavior of ε-automata and LCGA. One can see that in the game without a cooperation the multi-agent system converges to a steady-state with a corresponding average payoff of a team of players equal to 10, defined by the Nash point. In the game with a local cooperation a self-organization process can be observed. The system converges to a steady-state corresponding to playing the maximal price game, providing the maximum value of the average payoff received by the players and nearly equal to 30. In the game with a global cooperation a similar adaptive process can be observed.

The speed of convergence of the system is similar to the one observed in ε-automata games. Self-organizing features of CS-based systems open, however, new possibilities of using learning machines to solve real life problems.

4. APPLICATIONS: MAPPING AND SCHEDULING PROBLEMS

4.1. Problems of Mapping and Scheduling

Parallel and distributed computers built today consist of thousands of processors which need to coordinate their work. It is hard to imagine centralized control of such systems; therefore efficient algorithms for such control are needed. In this

section we consider two problems from the area of parallel and distributed processing, namely mapping and scheduling problems. These problems have the key influence on the performance of parallel and distributed systems. They both are known to be NP-complete and searching for effective heuristics to solve them is a subject of current research in the area. Recently proposed heuristics of mapping and scheduling are list scheduling algorithms [9], the dominant sequence clustering [37], or the dynamic critical path algorithm [15]. Some of them use a heuristic search based on simulated annealing [7], genetic algorithms [14], or neural networks [18]. Some insight into currently used methods of mapping and scheduling can be found in the recent literature [1, 2]. Most known heuristics are *serial* algorithms which are executed on sequential machines. A new and perspective line of research in the area is developing *parallel* algorithms of mapping and scheduling [3]. Constructing parallel mapping and scheduling algorithms results in a decreasing complexity of the algorithms. To develop *parallel and distributed* algorithms of mapping and scheduling we use the methodology of competitive multi-agent systems studied in previous sections.

Models of parallel programs and parallel computers which we accept in the paper are oriented on MIMD machines and are formulated as follows. A multiprocessor system is represented by an undirected unweighted graph $G_s = (V_s, E_s)$ called a *system graph*. V_s is the set of N_s nodes of the system graph representing processors with their local memories of a parallel computer of MIMD architecture. E_s is the set of edges representing bidirectional channels between processors and defines a topology of the multiprocessor system. If u and v are two vertices in G_s, (u, v) will denote the arc from u to v. The *distance* $d(u, v)$ between u and v is the length of the shortest path in G_s, connecting u and v, and is measured in a number of hops between u and v. Two processors corresponding to vertices u and v, respectively, are called *neighbor* processors if the distance $d(u, v) = 1$. It is assumed that all processors have the same computational power and a communication via the links does not consume any processor time.

A parallel program is represented by either a weighted directed acyclic graph called a *precedence task graph* (scheduling problem) or a weighted undirected graph (mapping problem) $G_p = \langle V_p, E_p \rangle$ with a set V_p of N_p nodes and a set E_p of edges (see Fig. 8). V_p is the set of N_p nodes of the graph representing elementary tasks, which are indivisible computational units. Weights b_k of the nodes describe the processing time needed to execute a given task on any processor of a given multiprocessor system. E_p is the set of edges of the task graph describing the communication pattern (connectivity) between the tasks. For the precedence task graph there exists a precedence constraint relation between the tasks k and l if the output produced by task k has to be communicated to the task l. Weights a_{kl} of the edges describe a communication time between pairs of tasks k and l, when they are located in neighbor processors. If the tasks k and l are located in processors corresponding to vertices u and v in G_s, then the communication delay between them will be defined as $a_{kl} * d(u, v)$.

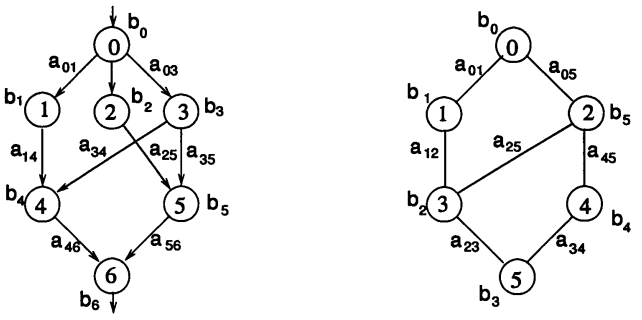

FIG. 8. Two models of parallel programs: directed (a) and undirected (b) program graphs.

The *mapping problem* can be formulated as the problem of seeking a mapping function θ that minimizes the *cost function* $C(\theta)$ describing total cost of processing and communication of a given parallel program in a given parallel architecture. The purpose of *scheduling* is to distribute tasks of a parallel program among the processors in such a way that precedence constraints are preserved and the *response time T* (the execution time) is minimized.

4.2. Mapping Problem

4.2.1. Dynamic Mapping. Current results concerning mapping communicating processes of a parallel program into parallel architectures show [12, 31] that applying static mapping algorithms raises the efficiency of using parallel computers when characteristics of programs do not change during their execution. However, if characteristics of the programs change in time, static policies cannot be efficiently applied. For this reason, a growing interest in developing algorithms and environments enabling the application of dynamic mapping [6, 17, 36] policies can be observed.

We propose an approach to dynamic mapping based on a multi-agent interpretation of a parallel program migrating in a parallel system environment to search an optimal allocation of program modules in a topology of a parallel system. To find multi-agent system strategies of migration in the system graph we use ε-automata and LCGA games models presented in the previous section.

Let θ be a mapping function from the vertex set of the program graph to the vertex set of the system graph and Θ the set of all mapping functions, i.e., $\Theta = \{\theta: V_p \to V_s\}$. We suppose that a local cost function $C(k, \theta)$ is defined for the kth program node mapped to the system graph. This function is defined as

$$C(k, \theta) = 0.5 \sum_{l=1}^{r_k} a_{kl} * d_{\min}(\theta(k), \theta(l)) + \sum_{n=1}^{n_i} b_n \qquad (11)$$

and describes a cost of communications of the kth program node with its neighbor program nodes and the computational load of a system node to which the kth program node was mapped, where r_k is the number of neighbor program nodes of the kth node; $d_{\min}(\theta(k), \theta(l))$ is a minimal number of hops

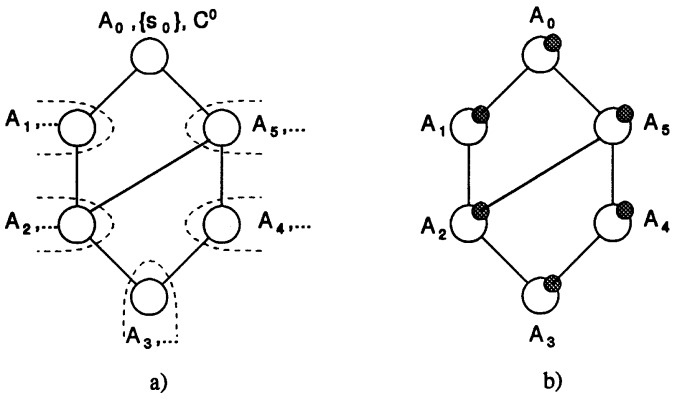

FIG. 9. Collection of agents assigned to program graph nodes (a), and a program graph interpreted as a multi-agent system (b).

between system nodes $\theta(k)$ and $\theta(l)$ where neighbor program nodes k and l are located, respectively, and n_i is a number of program nodes located in the system node $\theta(k)$.

The problem of static mapping can be formulated as the problem of seeking a mapping function $\theta \in \Theta$ that minimizes the total cost function $C(\theta)$ defined as a sum of the local cost functions (11), i.e.,

$$\min_{\theta \in \Theta} \left(C(\theta) = \sum_{k \in V_p} C(k, \theta) \right). \tag{12}$$

The dynamic mapping formulation of the problem can now be easily obtained due to locally defined cost functions and a multi-agent interpretation of the mapping problem. We assume that a collection of agents is assigned to nodes of the program graph in such a way that one agent is assigned to one program node (see Fig. 9). An agent A_k has a number of actions s_k which influence a local cost function C^k of a program node attached to the agent. If nodes of the program graph together with agents attached to them are placed in some way, e.g., randomly into the system graph, the agents' actions can be interpreted in terms of possible moves of the agents in the system graph (see Fig. 10).

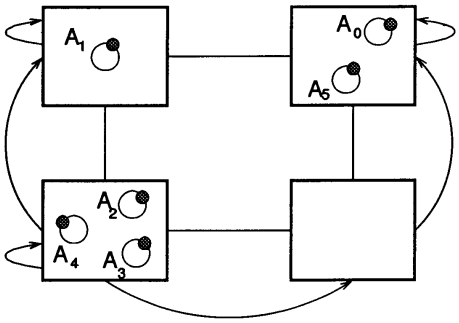

FIG. 10. Agents assigned to program nodes and mapped into a system graph play a game using actions: *do not migrate* or *migrate* to one of processors remoted on a distance of one hop. Only actions of agents A_2, A_3, and A_4 are shown.

The *dynamic mapping algorithm* can be specified in the following way:

- agents assigned to modules of a parallel program and located in system graph nodes are considered as players taking part in a game with limited interaction; each agent-player has $r + 1$ actions interpreted as follows: *do not migrate* or *migrate* to one of r nearest neighbor system nodes, where r is the degree of the system graph. Taking an action by a player corresponds to the player simulating the action, not a physical move;
- each player has a local cost function (11) describing the communications and computational costs; the function depends on the action of a given player and the actions of a limited number of neighbors;
- the objective of each player is to minimize its local cost function; a player maintains a local communication with neighbor players;
- the objective of the game is to find the optimal (according to (12)) directions of the migration of the program graph in the system graph while each player can move at the distance of a predefined number of hops;
- after a predefined number of games a migration of the program nodes (together with the agent-players) is performed; the game starts again.

In the case of a static program described above mapping algorithm can be considered as a parallel and distributed static mapping algorithm. However, we will show below that for parallel programs changing dynamically during their execution, the considered mapping algorithm is effectively able to perform dynamic mapping.

4.2.2. Dynamic Mapping with ε-learning Automata. In this section we present results of experiments with the dynamic mapping algorithm specified above and implemented with use of ε-learning automata. We do not make any specific assumptions about a topology of a parallel system, but in all experiments presented in the paper and concerning the mapping problem the target parallel machine contains eight processors connected according to cube topology. To model behaviors of dynamically changing programs we assume that programs randomly change their characteristics during execution. For the purpose of this experiment we assume that a program is represented by a sequence of three programs (Fig. 11): a grid4 $\times$ 4 with parameters $a_{kl} = 10$ and $b_k = 2$, a complete graph consisting of 16 modules with parameters $a_{kl} = 2$ and $b_k = 10$, and a ring16 with parameters $a_{kl} = 2$ and $b_k = 10$. All experiments have been conducted on a PC system.

The purpose of the first experiment was to define an optimal static mapping of each of the three programs from Fig. 11, assuming that each of them is a static program. We assume that before loading a static program into a parallel system, a static mapping algorithm running on a host is used to determine a plan of optimal allocation of program modules according to a global mapping criterion (12). For this purpose we use a standard GA-based static mapping algorithm. Figure 12 shows

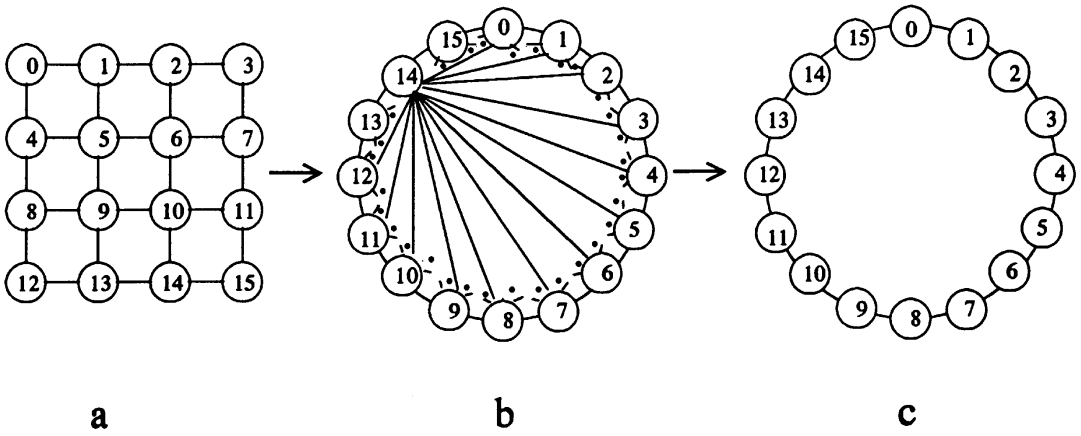

FIG. 11. Behavior of a dynamically changing program represented by a sequence of three program graphs: grid4 × 4 (a), complete graph 16 (b), and ring16 (c).

three typical runs of the static mapping algorithm for a program represented by the grid4 × 4 (Fig. 11a). Found optimal (or suboptimal) mapping corresponds to the value of $C(\theta)$ of the global criterion (12) equal to 174. For standard setting parameters of GA, such as a population size $n = 80$, $p_k = 0.9$, and $p_m = 0.005$, the algorithm finds an optimal mapping after more than 100 generations. Optimal mapping found for program graphs from Fig. 11b and 11c are characterized by values $C(\theta) = 544$ and $C(\theta) = 188$, respectively.

To define optimal values of parameters of ε-learning automata used in the dynamic mapping algorithm a number of experiments have been conducted with use of a program graph presented in Fig. 11a. The program is represented by the grid4 × 4 consisting of 16 nodes. The purpose of each such experiment was to find an optimal static mapping of a parallel program in a target architecture, using the dynamic mapping algorithm specified earlier. A program graph is initially randomly mapped into a parallel system which results in the initial physical allocation of the program with the cost $C(\theta) = 480$ (see Fig. 13). It is assumed that the execution of the program begins, but at the same time the mapping algorithm starts its work (game of ε-automata) monitoring the system and searching optimal migration strategies for program modules. One can see that during the first 1000 games, the system is able to find directions of program modules migration in a fully distributed way, providing a decrease in the cost allocation. After 1000 games a physical migration of the program modules is performed on a distance of one hop, which results in a new, better allocation with $C(\theta) = 220$, and the dynamic mapping algorithm continues to work. After three migrations of the program graph, near optimal mapping is found with $C(\theta) = 176$. The presented algorithm of static mapping is fully parallel and distributed. A number of conducted experiments shows that optimal parameters of ε-learning automata are $h = 16$ and $\varepsilon = 0.1$.

Figure 14 presents results of experiments with the dynamic mapping algorithm implemented with the use of ε-learning automata. It is supposed that a parallel program dynamicallychanges its computation requirements and a communication pattern (Fig. 11) during an execution.

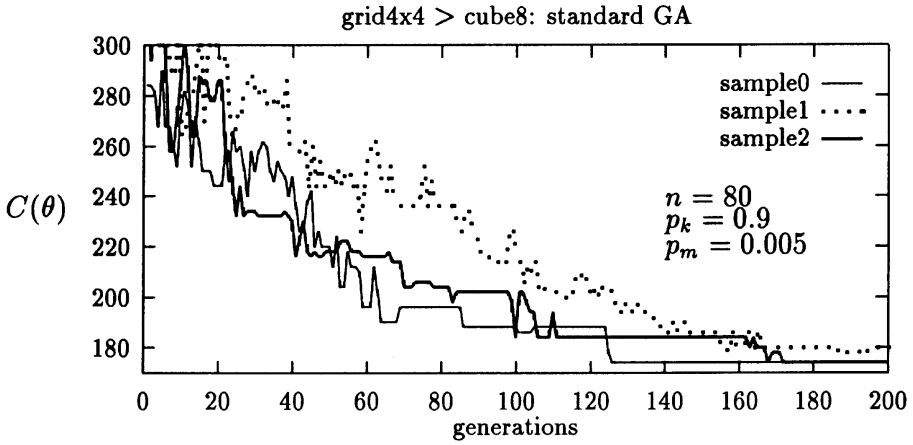

FIG. 12. Static mapping with use of standard GA.

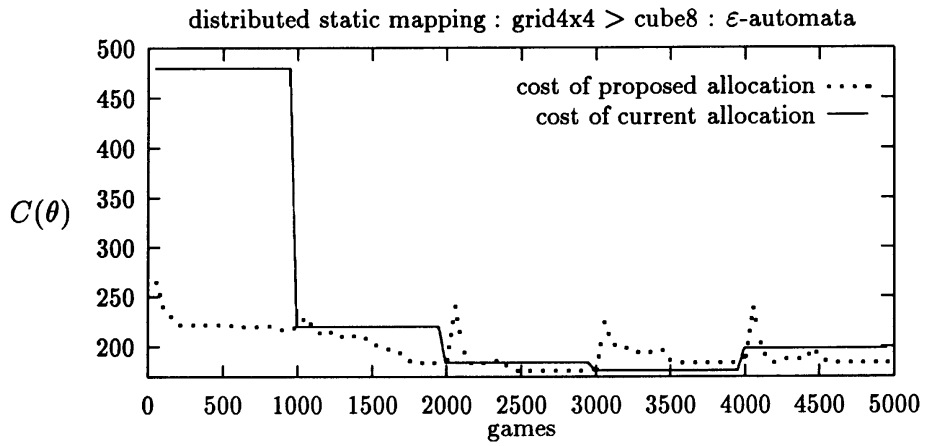

FIG. 13. Distributed static mapping with ε-learning automata.

The program represented initially by the grid4 $\times$ 4 from Fig. 11a is randomly mapped into the target system, which results in the initial physical allocation of the program with the cost $C(\theta) = 584$. Migration of modules is performed after each 1050 games and results in improving the cost of mapping.

It is assumed that after 6300 games the parallel program changes its communication pattern of activity and computation load, which is modeled by the complete graph shown in Fig. 11b. It is clear that the previously found allocation of the program is not optimal now, and the value of the cost function $C(\theta)$ is now equal to 656. The distributed dynamic mapping algorithm starts immediately to search for a new optimal allocation as it was shown earlier, without the need of any central synchronization.

After 11,550 games the parallel program changes its communication activity again, as modeled by the graph from Fig. 11c. The behavior of the distributed dynamic mapping algorithm is similar to that described above. The algorithm is able to catch the changes of the program graph parameters without any central synchronization and continues searching for an optimal mapping in a fully distributed way.

4.2.3. Dynamic Mapping with LCGA. In this section we use the competitive coevolutionary multi-agent system with LCGA implementation to solve the dynamic mapping problem. We conduct experiments in the same order as in the previous section, with the same pair of program and system graphs, to define the optimal parameters of LCGA implementing the dynamic mapping algorithm. We can see (Fig. 15) that the optimal static mapping of the same quality as in the case of applying ε-automata is found by a distributed static mapping algorithm for the values of the LCGA parameters: $n = 35$, crossover and mutation probabilities, respectively, equal to $p_c = 0.6$ and $p_m = 0.01$. Migration of program modules was performed after 30 generations of GA, which corresponds to the same number of games as in the experiment with ε-automata based mapping algorithms.

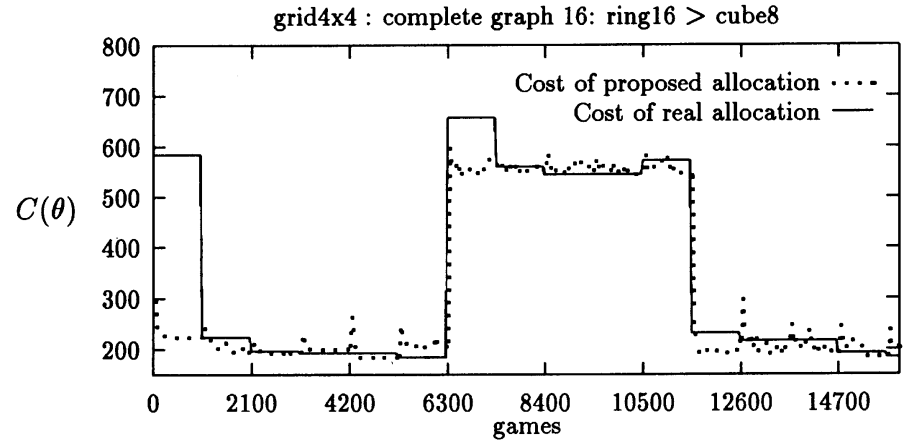

FIG. 14. Distributed dynamic mapping with ε-learning automata.

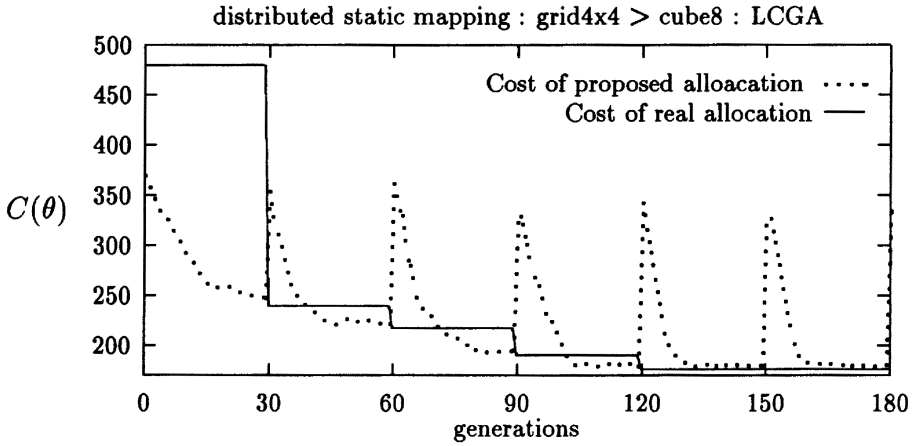

FIG. 15. Distributed static mapping with LCGA.

Figure 16 shows a performance of the dynamic mapping algorithm implemented by LCGA for a parallel program dynamically changing in time as shown in Fig. 11. It was assumed that the parallel program changes its communication pattern of activity and computation load after 150 and 300 generations of GA, respectively.

One can see that the performance of the dynamic mapping algorithm implemented by LCGA is similar to that implemented by ε-learning automata. However, the main difference between these two implementations is a cost of execution of both algorithms, measured by a volume of communication between agent-players. In the case of the implementation of the algorithm with the use of ε-learning automata, the automata should communicate after each game. In the case of the implementation with the use of LCGA the communication between players is less frequent and it occurs only after each generation of GA, i.e., after n games. Assuming that a communication delay T_{com} between two adjacent processors of a parallel system is described by a *linear communication model* [30] as

$$T_{\text{com}} = \beta + V \cdot \tau, \tag{13}$$

where β is a start-up cost, τ is the propagation time of a unit length message, and V is the message length, we can evaluate a communication cost of, e.g., 35 games (which corresponds to one subpopulation of LCGA with $n = 35$) between two players located in adjacent processors, in a game with ε-automata and LCGA implementations. Assuming a size of a message length (action of a player) sent by a player in a single game as 1 byte, we can calculate communication costs $T_{\text{com}}^{\varepsilon}$ and $T_{\text{com}}^{\text{LCGA}}$ as

$$T_{\text{com}}^{\varepsilon} = 35(\beta + 1 \cdot \tau) = 35\beta + 35\tau,$$

and

$$T_{\text{com}}^{\text{LCGA}} = \beta + 35\tau.$$

One can see that communication costs for ε-automata implementation of a game are much higher than for LCGA implementation, because of a high start-up component. The final evaluation of the performance of both algorithms depends on communication parameters of a target machine: in systems with small start-up values, learning automata-based mapping algorithms can be applied, otherwise GA-based mapping algorithms are preferable.

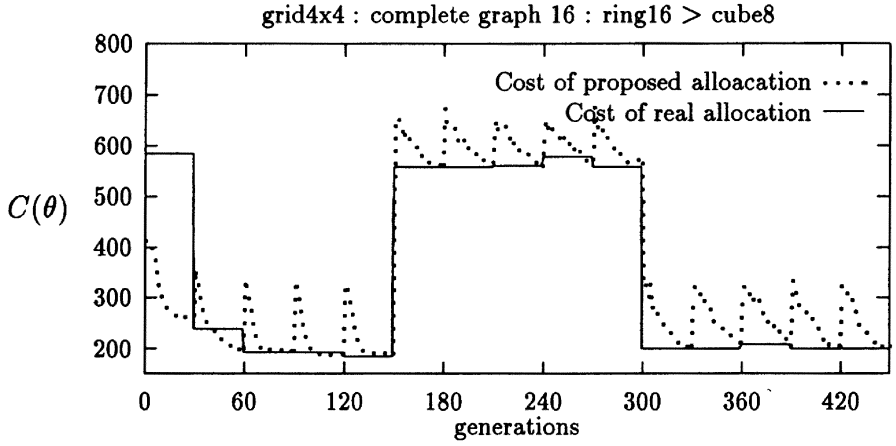

FIG. 16. Distributed dynamic mapping with LCGA.

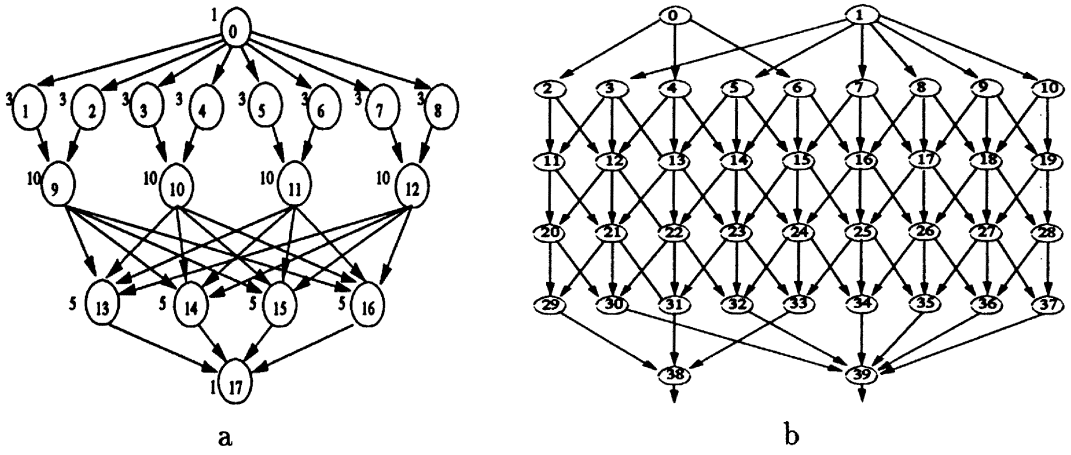

FIG. 17. Precedence task graphs used in experiments: the *graph18* (a) and the *graph40* (b).

4.3. Scheduling Problem

4.3.1. Task Scheduling with LCGA. We assume, as in the previous section, that a collection of agents is assigned to tasks of the precedence task graph. Each agent has a number of actions which influence a global function T describing the response time for a given scheduling problem. Such a model corresponds to a scheme with a global cooperation (see Section 2).

The scheduling algorithm essentially is similar to that described in the previous section. Agents, together with corresponding tasks randomly mapped into processors of a parallel system are considered as players taking part in a game which is implemented with LCGA. After a predefined number of games a physical migration of tasks in a parallel system is performed. Decreasing an execution time T of a program graph is expected in the result of migration. When only one, a final migration, is assumed, the algorithm can be considered as a distributed static scheduling algorithm. Results reported in this section concern this case.

In the first experiment we use a precedence task graph [8] presented in Fig. 17a. The graph has 18 tasks (the *graph18*) with computational costs marked in the graph and a communication cost of all edges equal to 1. The problem of scheduling the task graph into p-fully (p = 2, ..., 10) connected processors, and processors arranged in ring, cube, and deBruijn topologies (p = 8) is considered. The experiments have been conducted with the following values of LCGA parameters: n = 30, p_k = 0.6, p_m = 0.01. Results have been compared with ones obtained with GA (n = 100, p_k = 0.6, p_m = 0.01), and the list scheduling (LS) algorithm HLFET [9]. Table III summarizes the results.

In all conducted experiments, except the one for a cube topology, all three algorithms find the same response time. For the cube topology GA and LS algorithms find a schedule with T = 26, while LCGA finds a better schedule with T = 25. Figure 18a shows the found solution represented in a form of a Gantt chart showing the allocation of tasks on processors and the times when a given task starts and finishes its execution.

In the next experiment we use a precedence task graph [25] shown in Fig. 17b and consisting of 40 tasks. The precedence task graph (the *graph40*) has the same computational cost of all tasks equal to 4 and the same communication cost of all edges of the graph equal to 1. Table IV shows a response time T found by the algorithms.

LCGA and GA scheduling algorithms find the same response times in all experiments. In several cases results are better than ones produced by LS algorithms. Figure 18b shows a solution in a form of a Gantt chart for the *graph40* scheduled in the 5-fully connected processors. For this case, a schedule found by LCGA and GA algorithms defines the response time T = 40, which is better than that presented in [25], whose response time was equal to 43.

TABLE III
Response Time T for the *graph18*

	p-fully connected processors									Topology		
	p									Ring	Cube	deBruijn
Scheduling algorithm	2	3	4	5	6	7	8	9	10	8	8	8
LCGA	46	36	26	26	26	25	24	24	24	28	25	25

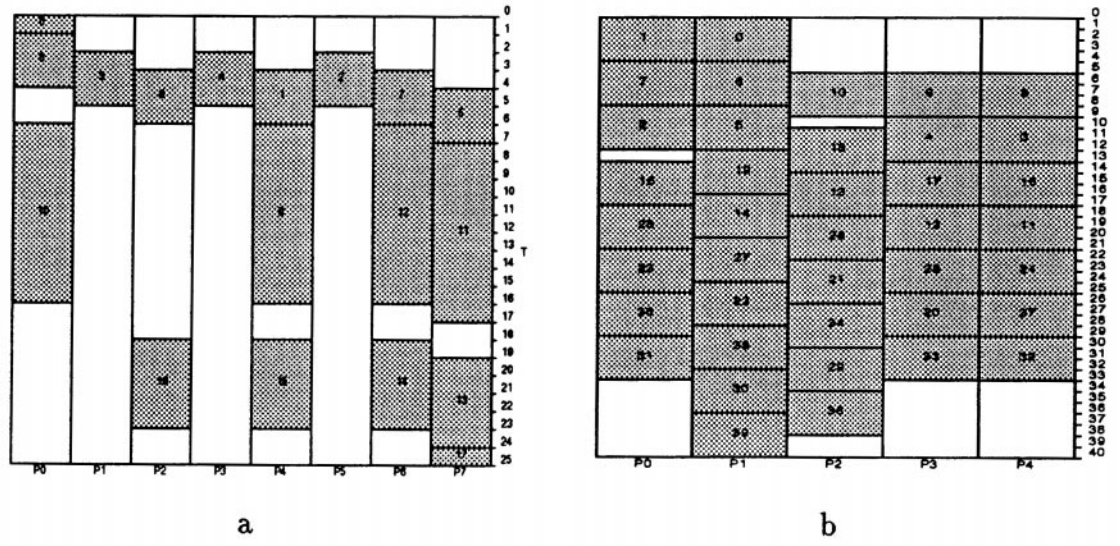

FIG. 18. Gantt charts for the precedence task graph from Fig. 17a scheduled in a cube (a), and for the precedence task graph from Fig. 17b scheduled in the 5-fully connected processors (b).

In the last experiment a random graph with 100 nodes (the *graph100* was used. A computational cost of each task was an integer random number between 3 and 7. Communication costs of edges were integer random numbers between 2 and 4. Table V summarizes results found with LCGA, GA, and LS algorithms.

One can see that for this larger graph the algorithms produce different results. In each case the best results are found by LCGA algorithms. GA gives better results than LS algorithms for fully connected systems and worse results for systems with ring, cube, and deBruijn topologies. An improvement of results found by LCGA is usually about a few percent and reaches about 10% for the case of scheduling in the 4-fully connected processors.

Figure 19 gives some insight into a complexity of both LCGA and GA algorithms. It shows a run of both algorithms as a function of a number of evaluations of T, which are necessary for a convergence of the algorithms for the case of scheduling of the random *graph100* in the 4-fully connected processors. The figure presents runs of three samples of each algorithm converging to their solutions. One can see that LCGA converges approximately two times faster than GA, and finds a solution of much better quality.

4.4. Task Scheduling with Classifier Systems: Negotiation Model

In this section we examine a possibility of using GA-based learning machines (see Section 3.4) to build a scheduler. As in the previous section we interpret a parallel program represented by a precedence task graph as a multi-agent system. Each agent is a CS learning machine and it is responsible for local mapping decisions concerning a given task. Each CS has a number of classifiers describing rules and corresponding to them decisions which are used in a process of an iterated game. Classifiers will describe accepted attributes and basic heuristics and will influence a performance of the proposed approach.

It is assumed that a conditional part of a classifier will describe situations (for details, see [28]) which can be recognized by an agent A_k located in some system node. Objects recognized by an agent are some specific tasks of a program graph. An object is considered "close" to an agent if the distance to it in a system graph is less than or equal to a predefined number of hops; otherwise, it is considered "far."

Objects recognized by an agent are, in particular,

- a task predecessor whose data arrive last

TABLE IV
Response Time *T* for the *graph40*

	p-fully connected processors									Topology		
	p									Ring	Cube	deBruijn
Scheduling algorithm	2	3	4	5	6	7	8	9	10	8	8	8
LCGA	80	57	45	40	35	33	32	29	29	34	33	33
List alg.	81	57	47	42	38	34	32	29	29	38	35	36

TABLE V
Response Time *T* for the *graph100*

	p-fully connected processors								Topology		
	p								Ring	Cube	deBruijn
Scheduling algorithm	2	3	4	5	6	7	8	9	8	8	8
LCGA	247	180	158	153	151	152	151	152	186	173	168
GA	254	196	169	159	154	153	153	154	198	180	174
List alg.	264	197	175	168	158	158	158	158	194	176	171

- tasks predecessors and tasks successors with the highest processing time
- tasks from the same level of a program graph
- not immediate task predecesors and successors.

These objects can be classified as being all "far," as some of them being "close," or all of them being "close." Some other attributes of a situation in a system graph can be recognized, such as a delay of a given task to be processed on a processor or to a communication channel and a total delay of tasks located in a given node to a processor and to communication channels. These delays can be classified as "acceptable" or "not acceptable" by a comparison with similar characteristics of tasks located in neighboring system nodes.

An action part of a classifier will describe potential actions (moving on some distance in a system graph) which can be taken by an agent in a given situation. The list contains 16 actions, some of which are given below:

- *a0*, stay in a processor where you are located (do not move)
- *a3*, move to a processor where your randomly chosen successor is located
- *a5*, move to processor where a predecessor with a minimal computation cost is located
- *a11*, move to the lowest loaded processor
- *a13*, move to a processor where your "brothers" (tasks from the same level of the precedence graph) are not located
- *a14*, move to a VIP processor, i.e., a processor where a predecessor whose data arrive last is located.

Conditional and action parts of a classifier contain information concerning the condition and the action represented by the classifier, coded with binary substrings, and a don't care symbol # (only conditional part). For example, a classifier

$$\langle 1\ 0\ \#\#\ 0\ 1\ \#\#\ \ldots\ 1\ 0\ \#\#\rangle{:}\langle 0\ 0\ 1\ 1\rangle$$

can be interpreted in the following way:

IF (all task predecessors in which data arrives last are "close"
AND task predecessors with the highest processing time is not important where (# symbol)
AND task—"brothers" are not important where
...
AND total waiting communication time of tasks located in the same processor is greater than the total waiting time in the neighbor processor
AND total waiting processing time in the processor—its value not important
THEN move to the processor of your randomly selected successor.

In contrast to the model of a game considered in previous sections we assume that a single game is conducted as a sequence of moves (actions) of agents, i.e., at a given moment of time only one agent takes an action. The order of taking

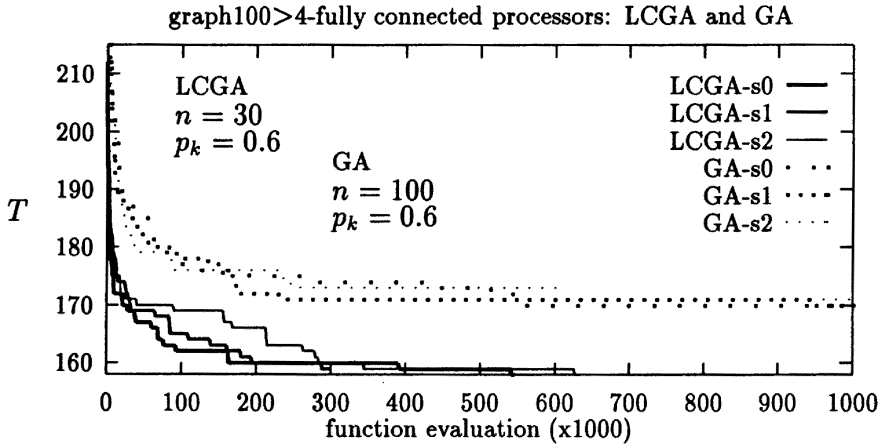

FIG. 19. Comparisons of performance of LCGA and GA scheduling algorithms.

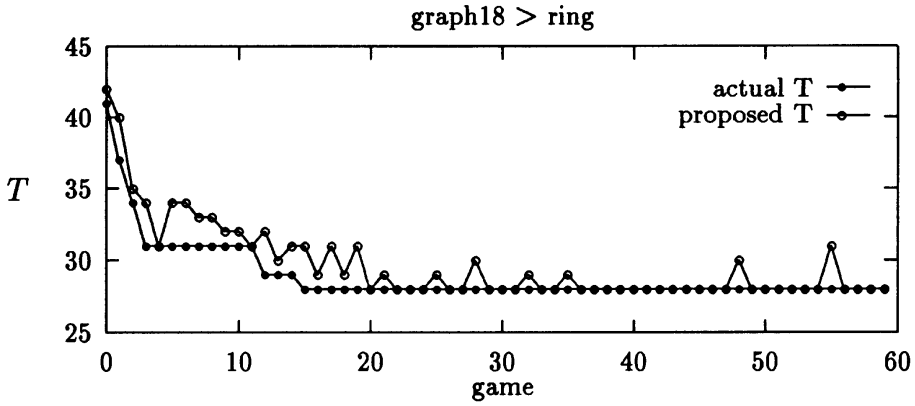

FIG. 20. Performance of CS-based negotiation scheduler.

decisions by agents is defined by their number in a precedence graph. A single game is completed in N_p moments of time. A game consists of G single games g, i.e., $g = 1, 2, \ldots, G$. We introduce into the game a new option: a player can withdraw his action in the case of failure, i.e., increasing T as a result of his action. We call such a model of the game a game with *negotiation*.

Figure 20 shows a performance of the scheduler applied to a precedence task graph from Fig. 17a scheduled in a ring multiprocessor topology. One can see that the algorithm very quickly finds a solution with $T = 28$, rejecting in the negotiation process solutions which increase T. Figure 21 gives some insight into a process of searching a solution during the game, showing change of frequencies *freq* of some actions used by *CS* in a process of the game.

We can observe a collective behavior of a set of active actions which contribute to the process of searching for a solution, while the other actions serve as a background of the process of searching. This active set is created by the actions *a11*, move to the lowest loaded processor; *a13*, move to a processor where "brothers," i.e., tasks from the same level of a program graph which have the same predecessor are not located; and *a14*, move to a VIP processor. When the algorithm approaches the solution, an increasing importance of the action *a0*, stay in a processor where you are located, can be observed.

We believe that applying GA-based learning machines to scheduling problems is a very promising direction of research, which will result in discovering new effective heuristics of scheduling.

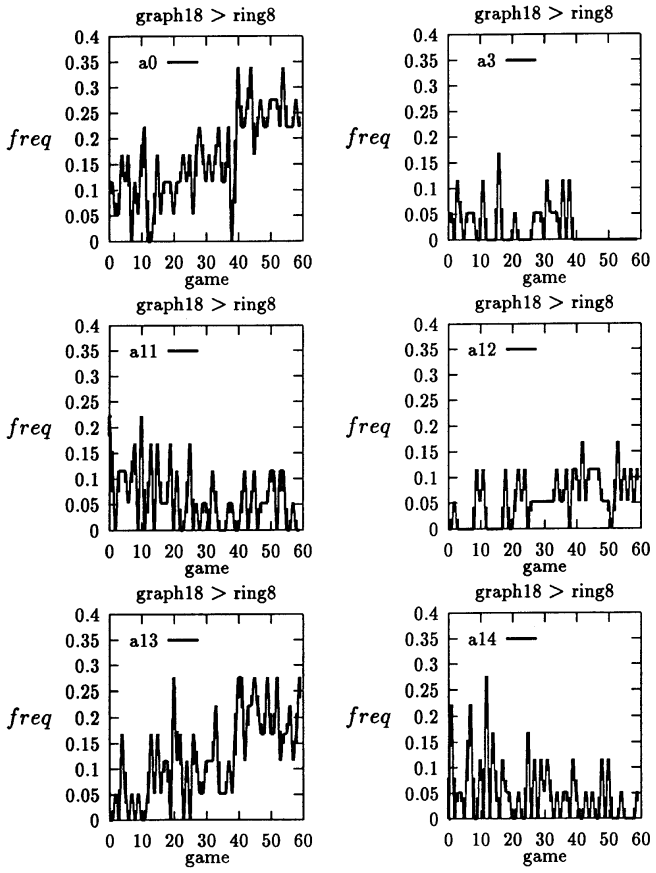

FIG. 21. Frequencies of selected actions of CS-based scheduler.

5. CONCLUSIONS

We have proposed in this paper a new paradigm for a parallel and distributed evolutionary computation based on the model of noncooperative games with limited interaction. We addressed the problem of a global behavior of a team of players, measured by a value of the average payoff received by the team in an iterative game. We showed the rules of a local interaction between players providing a global behavior of the system. To implement a competitive behavior of players we proposed three parallel and distributed schemes with an evaluation of local fitness functions of players taking part in iterated games. Conducted experiments have shown that all three systems are capable of evolving a global behavior, in particular, if rules of only local cooperation between players are preserved. Behavior of the systems in its main points is similar to that predicted by game theory with its concept of a Nash equilibrium point.

We have applied competitive coevolving multi-agent systems to develop parallel and distributed algorithms to solve effectively two problems from the area of parallel and distributed processing: the dynamic mapping problem and scheduling problem. We believe that the presented results give rise to believe that the developed model can serve as a useful metaphor for distributed decision-making and distributed control in real life systems. A recently conducted study [29] shows that the paradigm introduced in the paper can also be effectively applied in the area of distributed function optimization.

ACKNOWLEDGMENT

The author thanks the anonymous reviewers for their constructive comments and suggestions which improved the quality of the presentation.

REFERENCES

1. Ahmad, I. (Ed.). Special Issue on Resource Management of Parallel and Distributed Systems with Static Scheduling: Challenges, Solutions and New Problems. *Concurrency: Practice and Experience* **4,** 5, (1995).
2. Ahmad, I. (Ed.). Special Issue on Resource Management in Parallel and Distributed Systems with Dynamic Scheduling: Dynamic Scheduling. *Concurrency: Practice and Experience* **7,** 7, (1995).
3. Ahmad, I., and Kwok, Y. A parallel approach for multiprocessing scheduling. *9th Int. Parallel Processing Symposium,* Santa Barbara, CA, April 25–28, 1995.
4. Axelrod, R. The evolution of strategies in the iterated prisoners' dilemma. In Davis, L. (Ed.). *Genetic Algorithms and Simulated Annealing.* Pitman, London, 1987.
5. Błażewicz, J., Ecker, K. H., Schmidt, G., and Węglarz, J. *Scheduling in Computer and Manufacturing Systems.* Springer-Verlag, Berlin/New York, 1994.
6. Candlin, R., and Philips, J. The dynamic behaviour of parallel programs under process migration. *Concurrency: Practice and Experience* **7,** 7 (1995), 591–613.
7. Chang, H. W. D., and Oldham, W. J. B. Dynamic task allocation models for large distributed computing systems. *IEEE Trans. Parallel Distrib. Systems* **6,** 2 (Dec. 1995).
8. El-Rewini, H., and Lewis, T. G. Scheduling parallel program tasks onto arbitrary target machines, *J. Parallel Distrib. Comput.* **9** (1990), 138–153.
9. El-Rewini, H., Lewis, T. G., and Ali, H. H. *Task Scheduling in Parallel and Distributed Systems.* PTR Prentice Hall, New York, 1994.
10. Fogel, D. B. *Evolutionary Computation. Towards a New Philosophy of Machine Intelligence.* IEEE Press, New York, 1995.
11. Fox, G. C., Johnson, M., Lyzenga, G., Salmon, J., and Walker, D. *Solving Problems on Concurrent Processors.* Prentice Hall, New York, 1988.
12. Gaudiot, J. L., and Pi, J. I. Program graph allocation in distributed multicomputers. *Parallel Comput.* **7** (1989).
13. Goldberg, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison–Wesley, Reading, MA, 1989.
14. Hou, E. S. H., Ansari, N., and Ren, H. A genetic algorithm for multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Systems* **5,** 2 (Feb. 1994).
15. Kwok, Y. K., and Ahmad, I. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. Parallel Distrib. Systems* **7,** 5 (May 1996), 506–521.
16. Levy, R., and Rosenschein, J. S. A game theoretic approach to distributed artificial intelligence and the pursuit problem. In Werner, E., and Demazeau, Y. (Eds.). *Decentralized A.I.-3.* Elsevier, Amsterdam, New York, 1992.
17. Melab, N., Devesa, N., Lecouffe, M. P., and Toursel, B. Adaptive load balancing of irregular applications a case study: IDA applied to the 15-puzzle problem. *IRREGULAR'96.* Lecture Notes in Computer Science, Vol. 1117, Springer-Verlag, Berlin/New York, 1996.
18. Meyer, J. W. Self-organizing processes. In Buchberger, B., and Volkert, J. (Eds.). *CONPAR94—VAPPVI.* Lecture Notes in Computer Science, Vol. 854, Springer-Verlag, Berlin/New York, 1992.
19. Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, Berlin/New York, 1992.
20. Mühlenbein, H., Gorges-Schleuter, M., and Kramer, O. Evolution algorithms in combinatorial optimization. *Parallel Comput.* **7** (1988), 65–85.
21. Narendra, K. S., and Thathachar, M. A. L. *Learning Automata—An Introduction.* Prentice Hall, Englewood Cliffs, 1989.
22. Ordeshook, P. C. *Game Theory and Political Theory: An Introduction.* Cambridge Univ. Press, Cambridge, UK, 1986.
23. Potter, M. A., and De Jong, K. A. A cooperative coevolutionary approach to function optimization. In Davidor, Y., Schwefel, H.-P., Männer, R. (Eds.). *Parallel Problem Solving from Nature—PPSN III.* Lecture Notes in Computer Science, Vol. 866, Springer-Verlag, Berlin/New York, 1994.
24. Sastry, P. S., Phansalkar, V. V., and Thathachar, M. A. L. Decentralized learning of Nash equilibria in multi-person stochastic games with incomplete information. *IEEE Trans. Systems Man. Cybernet.* **24,** 5 (May 1994), 769–777.
25. Schwehm, M., and Walter, T. Mappign and scheduling by genetic algorithms. In Buchberger, B., and Volkert, J. (Eds.). *CONPAR94—VAPPVI.* Lecture Notes in Computer Science, Vol. 854, Springer-Verlag, Berlin/New York, 1994.
26. Seredynski, F. Homogeneous networks of learning automata. Tech. Rep. N 684. Institute of Computer Science PAS, Warsaw, 1990.
27. Seredynski, F. Loosely coupled distributed genetic algorithms. In Davidor, Y., Schwefel, H.-P., and Männer, R. (Eds.). *Parallel Problem Solving from Nature—PPSN III,* Lecture Notes in Computer Science, Vol. 866, Springer-Verlag, Berlin/New York, 1994.
28. Seredynski, F., Task Scheduling with Use of Classifier Systems. In *Evolutionary Computing.* Lecture Notes in Computer Science, Vol. 1305 (D. Corne and J. L. Shapiro, Eds.), Springer-Verlag, Berlin/New York, 1997.
29. Seredynski, F., Bouvry, P., and Arbab, F. Parallel and distributed evolutionary computation with manifold, *Parallel Computing Technologies.* Lecture Notes in Computer Science, Vol. 1277 (V. Malyshkin, Ed.), Springer-Verlag, Berlin/New York, 1997.
30. Stout, Q. F., and Wagar, B. Intensive hypercube communication, prearranged communication in link-bound machines. *J. Parallel Distrib. Comput.* **10** (1990), 167–181.
31. Shirazi, B., Wang, M., and Pathak, G. Analysis and evaluation of heuristic methods for static task scheduling. *J. Parallel Distrib. Comput.* **10** (1990).
32. Tsetlin, M. L. *Automaton Theory and Modelling of Biological Systems.* Academic Press, New York, 1973.
33. Varshavsky, V. I., Zabolotnyj, A. M., and Seredynski, F. Homogeneous games with an associated exchange process. In *Engineering Cybernetics,* Vol. 15, No. 6, 1977, Scripta Technica, New York, 1978.

34. Warschawski, W. I. *Kollektives Verhalten von Automaten.* Akademie-Verlag, Berlin, 1978.
35. Whitley, D., and Starkweather, T. Genitor II: A distributed genetic algorithm. *J. Experiment. Theoret. Artif. Intell.* **2** (1990), 189–214.
36. Willebeek-leMair, M. H., and Reeves, A. P. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel Distrib. Systems* **4,** 9 (Sept. 1993), 979–993.
37. Yang, T., and Gerasoulis, A. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel Distrib. Systems* **5,** 9 (Sept. 1994), 951–967.
38. Yao, X., and Darven, P. J. An experimental study of N-person iterated prisoner's dilemma games. In Yao, X. (Ed.). *Progress in Evolutionary Computation.* Lecture Notes in Artificial Intelligence, Vol. 956, Springer-Verlag, Berlin/New York, 1995.

Received February 1, 1997; revised July 15, 1997; accepted August 25, 1997

FRANCISZEK SEREDYNSKI received his Ph.D. and M.S. degrees in computer science from the Leningrad Electrotechnical Institute in 1978 and 1973, respectively. He is currently an assistant professor at the Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland. His research interests include evolutionary computation techniques, multi-agent systems, and parallel and distributed processing. He was a visiting researcher at the Centre for Mathematics and Computer Science, Amsterdam, The Netherlands (1996), the International Computer Science Institute, Berkeley, USA (1995), and the Institut National Polytechnique de Grenoble, France (1991–1992). He has served on program committees for a number of international conferences in the areas of evolutionary computing and parallel processing.